Automatic Test Program Generation Method

Background of the Invention

The present invention relates to test program preparation methods and, more specifically, to a method for generating a test program used to test compilers under development.

Programs written in high level language are converted to executable forms by compilers. In the development of a compiler, test programs are prepared in order to verify its functions. Conventionally, such a test program is manually prepared by the designer of the compiler or an expert familiar with the language specification.

On the other hand, some approaches have also been proposed aimed at automatic generation of a test program by a computer. One of them is to select rules from prepared plural syntax generation rules by means of random numbers and generates a test program based on the selected rules as disclosed in Japanese Patent Laid-open No. 1-220044. Another approach is to check the grammatical coverage of an existing test program, output the non-covered grammars as a reduced set of grammars and automatically generate an additional test program based on the reduced set of grammars as disclosed in Japanese Patent Laid-open No. 3-75840. In still another

approach, statement patterns are generated based on syntactic rules described in the Backus Naur form and the generated statement patterns are manually edited into a test program as disclosed in Japanese Patent Laid-open No. 6-214775.

Any of these automatic test program generation methods automatically generate single statements in a test program.

Summary of the Invention

To remove functional defects from a compiler, it is necessary to test the compiler by using many prepared test programs. However, architecturally complicated recent computers require complicated compiler functions. To test all functions of a compiler, an enormous amount of man-power is needed if the test programs are manually prepared as conventional. Therefore, functional verification may not satisfactorily be done before the compiler is shipped.

To cope with this situation, automatic test program generation methods have so far been proposed as mentioned above. However, these methods are to generate single statements. Hard-to-find serious functional problems are generally caused by structurally complicated programs. They do occur not within statements but across plural statements if many conditions are met. Such functional problems are difficult to find by the conventional automatic generation

methods designed to mainly generate single statements.

According to an automatic test program generation method by the present invention, a plurality of partial program descriptions (sub-procedural descriptions, called program cells, each of which may form an element of a test program) are selected by a random number-used method and combined into a test program.

Brief Description of the Drawings

FIG. 1 shows an example of a configuration of a test program generator;

FIG. 2 shows an example of a configuration of a compiler test system;

FIG. 3 shows an example of a configuration of a basic cell;

FIG. 4 shows an example of a configuration of a select cell;

FIG. 5 shows an example of a configuration of a repeat cell;

FIG. 6 shows an example of a configuration of a function cell;

FIG. 7 shows an example of a process sequence by a program cell combining unit;

FIG. 8 shows an example of a cell management table;

FIG. 9 shows an example of a cell description table;

FIG. 10 shows an example of a selected-cell table; and

FIG. 11 shows an example of a declaration/definition (declaration and definition) cell.

Description of the Preferred Embodiments

The following will describes embodiments of the present invention with reference to the drawings.

FIG. 1 shows an example of a configuration of a test program generator 100 that automatically generates a test program for a compiler by implementing an automatic test program generation method of the present invention. The test program generator 100 shown as the present embodiment is realized, for example, by software that operates on a personal computer.

In FIG. 1, program cell information 110, stored in a memory unit of a computer, is a set of pieces of information, each of which describes part of the program. As described later in detail with reference to FIGS. 3, 4, 5 and 11, program cells (also denoted simply as cells) are components of the test program. Each cell is given a weight as specified by cell weight information 111. The cell weights are treated in such a manner that giving a higher weight to a program cell raises the probability of the program cell being selected when

specific program cells are selected from the program cell set by using random numbers. Based on random numbers generated by a random number generator 120, a program cell combining unit 130 selects program cells and combines the selected program cells to generate a test program 140.

Weights are given to such cells as those in which tested program bugs were found in the past and those in which bugs are likely to be found. This weighting can be changed on an each test basis.

FIG. 2 shows an example of a configuration of a compiler test system that incorporates the test program generator 100 described with FIG. 1. This system is implemented, for example, by software that operates on a personal computer. A test program 140, which is an output of the test program generator 100, is entered into a testing unit 220. In the testing unit 220, the test program is compiled (230) by the test target compiler. Then, an executable file obtained by the compilation is executed (240) and the result is stored as an execution result 260 in a storage unit.

The same test program 140 is also entered into an expected value generator 210. The expected value generator 210 can be implemented, for example, by using a compiler which has been proven through practical use. The test program is compiled to an executable file by the proven compiler and then

executed. An expected value 250 obtained by entering the test program 140 into the expected value generator 210 and the execution result 260 are entered into a result comparator 270. The comparison result is stored as a test result 280 in a storage unit of the computer.

Before a test program is generated as described later in detail with reference to FIG. 7, each cell is tested alone to check the integrity of the cell itself. In addition, although the above-mentioned test result of the test program is a result of executing a plurality of cells combined into one program, the test result is configured to indicate which cell is abnormal if any.

By using PADs, FIGS. 3 to 6 and 11 show program cells configured as proposed by the present invention. In the following description, the program cells are classified into four types: basic cells, control cells, function cells and declaration/definition cells.

FIG. 3 is a PAD showing an example of a configuration of the basic cell. The basic cell is a program cell where "Definition of Variables etc. and Setting of Initial Values" 320, "Executable Statements" 330 and "Displaying and Storing of Execution Result" 340 are described in the order of execution. Further, "Description 1 Necessary for Unit Operation" 310 and "Description 2 Necessary for Unit

Operation" 350 are included which are necessary to separately compile/execute the basic cell. By means of optional specification at compilation, it is possible to control "Description 1 Necessary for Unit Operation" 310 and "Description 2 Necessary for Unit Operation" 350 so as to compile and operationally check the basic cell as a single unit.

For example, "Description 1 Necessary for Unit Operation" 310 may be written as follows:

```
#ifdef_CELL_COMPILE_CHECK_
   void main(){
   #endif
```

Likewise, "Description 2 Necessary for Unit Operation" 350 may be written as follows:

```
#ifdef_CELL_COMPILE_CHECK_
    }
   #endif
```

Unless the token _CELL_COMPILE_CHECK_is defined, the cell cannot be compiled/executed as a single cell. To compile/execute a test program made of program cells combined, the token _CELL_COMPILE_CHECK_is not defined.

In "Definition of Variables etc. and Setting of Initial Values" 320, variables and others necessary to compile/execute the basic cell are defined and initialized.

In addition to variables, constants, macros, etc. can be defined.

"Displaying and Storing of Execution Result" 340 is described to present the execution result of the basic cell on a display unit connected to the computer and store it in a storage unit. By this, while a test program incorporating program cells is being executed, the test operator can grasp the test program's operational progress in detail.

FIGS. 4 and 5 are PADS for showing examples of configurations of control cells. In this embodiment, the control cells are classified into select cells as shown in FIG. 4 and repeat cells as shown in FIG. 5. Either cell incorporates "Description 1 Necessary for Unit Operation" 310 and "Description 2 Necessary for Unit Operation" 350. Therefore, by means of optional specification at compilation, it is possible to control "Description 1 Necessary for Unit Operation" 310 and "Description 2 Necessary for Unit Operation" 350 so as to compile and operationally check the control cell as a single cell.

FIG. 4 is a PAD showing an example of a configuration of the select cell. Like in a basic cell, "Definition of Variables etc. and Setting of Initial Values" 410 is described to define and initialize variables and others which are necessary to compile/execute the select cell on the computer.

"Selection" 420 is described so as to invoke "Executable Statements" 430 depending on conditions described therein. Each select cell can have "other program cells" 440 nested therein although it can also be executed with no other program cell nested therein. Like in a basic cell, "Displaying and Storing of Execution Result" 450 is described to present the execution result of the select cell on a display unit connected to the computer and store it in a storage unit.

FIG. 5 is a PAD showing an example of a configuration of a repeat cell. Like in a basic or select cell, "Definition of Variables etc. and Setting of Initial Values" 510 is described to define and initialize variables and others which are necessary to compile/execute the select cell on the computer. "Repeating" 520 is described to repeatedly invoke "Executable Statements" 530. Similar to a select cell, a repeat cell can have "Other Program Cells" 540 nested therein, too. "Displaying and Storing of Execution Result" 550, like in a basic or select cell, is described to present the execution result of the repeat cell on a display unit connected to the computer and store it in a storage unit.

FIG. 6 is a PAD showing an example of a configuration of a function cell. Similar to basic and control cells, each function cell can be executed alone. "Description 1 Necessary for Unit Operation" 310 and "Description 2 Necessary for Unit

Operation" 350 are incorporated. Therefore, by means of optional specification at compilation, it is possible to control "Description 1 Necessary for Unit Operation" 310 and "Description 2 Necessary for Unit Operation" 350 so as to compile and operationally check the function cell as a single cell. "Definition of Variables etc. and Setting of Initial Values" 610, like in basic and control cells, is described to define and initialize variables and others which are necessary to compile/execute the function cell on the computer. "Function" 620 is described to call a function defined in another place. "Displaying and Storing of Execution Result" 630, like in basic and control cells, is described to present the execution result of the function cell on a display unit connected to the computer and store it in a storage unit.

FIG. 11 is a PAD showing an example of a configuration of a declaration/definition cell. In the declaration/definition cell, structures which are to be shared by other program cells are declared, functions are defined and #include statements are described. When program cells are combined into a test program, one declaration/definition cell is always placed at the head.

FIG. 7 shows a process sequence performed by the program cell combining unit 130 of FIG. 1. The program cell information 110 comprising basic, control, function and

declaration/definition cells, shown respectively in FIG. 3 to FIG. 6 and FIG. 11, and the cell weight information 111 are taken in from storage units of the computer. Then, the program cell combining unit 130 selects program cells by using random numbers generated by the random number generator 120, and combine them to generate a test program 140. This process sequence will be described later in detail.

FIGS. 8, 9 and 10 show examples of tables which are to be formed in the main memory of the computer when the process sequence of FIG. 7 is implemented by computer software. FIG. 8 shows a cell management table 800. In an item number column 810, the entry number of each cell is stored. In a cell name column 820, each cell's name is stored. A cell type column 830 stores each cell's cell type: basic, control, function or declaration/definition.

In a cell weight column 840, a value given to each cell is stored, reflecting the probability of the cell being selected using a random number from those registered to the cell management table. For example, the selecting scheme may be designed in such a manner that the probability of a weight 2 program cell being selected is twice as high as that of a weight 1 program cell being selected. In a cell description table information column 850, information about a cell description table 900 is stored. The cell description table

information column 850 is divided into a pointer column 851 and a description length column 852. The pointer column 851 indicates the address of the first character of each cell's corresponding cell description in a cell description table 900 whereas the description length column 852 indicates the length of the cell description with the number of characters therein.

FIG. 9 shows an example of the cell description table 900. The cell description table 900 has an address row 910 and a character row 920. The address row 910 indicates the address of each character stored in the cell description table 900 whereas the character row 920 stores each character of each cell description. For example, consider cell No. 1 in FIG. 8. As the cell description table information 850 regarding the cell No. 1, 0 and 156 are respectively stored in the pointer column 851 and description length column 852. This means that the description of cell No. 1 starts at address 0 and ends at address 155 in FIG. 9. Note that any character stored therein, whether it is an alphanumeric, space, line feed symbol or something else, is assumed as necessary for cell description. Therefore, when cell No. 1 in FIG. 8 is to be executed, characters are read out from addresses 0 through 155 in FIG. 9.

FIG. 10 shows an example of a configuration of a

selected cell table 1000. The selected cell table 1000 has an execution order column 1010 and a selected cell pointer column 1020. The execution order column 1010 indicates what number entry each program cell, selected using a random number according to the process sequence shown in FIG. 7, is in the selected cell table (what number cell each selected cell is in the test program). The selected cell pointer column 1020 indicates what cell No. is attached in the cell management table 800 to each cell which is selected using a random number according to the process sequence shown in FIG. 7.

With reference to the tables shown in FIGS. 8 to 10, the following describes how the program cell combining unit implements the process sequence shown in FIG. 7.

In the process sequence shown in FIG. 7, the program cell information 110 and the cell weight information 111 in FIG. 1 are deployed into the cell management table 800 in FIG. 8 and the cell description table 900 in FIG. 9 by "Generation of Internal Tables" 710. "Creation of a List of Program Cells by Means of Random Number" 720 selects program cells from those registered with the cell management table 800 by using random numbers generated by the random number generator 120. The probability of each cell being selected is controlled based on the weight 840 given to the cell in the cell management table 800. For example, this probability control may be

designed in such a manner that the probability of a weight 3 program cell being selected is three times as high as that of a program cell given a cell weight of 1. Each selected program cell is registered with the selected cell table 1000. That is, the execution order column 1010 stores the numbers of the selected cells in the order of selection whereas the selected cell pointer column 1020 stores the item Nos. given in the cell management table 800 to the selected program cells.

In the order of registration with the selected cell table 1000, the following process is performed for each program cell registered with the selected cell table 1000 (730). It is judged whether the program cell preceding the current cell is a basic/function cell or a control cell (740) and, if the preceding cell is a basic cell or a function cell, the current cell is concatenated to the preceding cell (750). If the preceding cell is a control cell, the current program cell is nested into the control cell (760).

Then, variables are made sharable among the program cells which are arranged to constitute a test program according to the above-mentioned procedure (770). Making a variable sharable means making its variable name common among the cells so that an operation result in a program cell can be reflected in the subsequent operation in a later cell. Although variables are defined uniquely in program cells,

making variable sharable herein allows data exchange among the program cells.

According to the process sequence mentioned so far, it is possible to automatically generate a test program.

Since the program cells are classified into types and the cells to be executed and their order of execution are determined based on the type of each program cell, it is possible to generate a variety of test programs necessary to test a compiler.

The above-mentioned embodiment can automatically generate a satisfactory test program for testing a compiler by selecting program cells in the order of execution from a plurality of program cells by using random numbers based on cell weights assigned to the program cells. The selected cells are combined into a test program. Further, since selection tendency of the selected program cells can be changed by adjusting cell weights, it is possible to direct the test program to specific defects which are liable to occur in the compiler. Accordingly, this automatic test program generation can reduce the man-hour required to generate a test program and makes it possible to test many aspects of a compiler.

According to the present invention, it is possible to automatically generate a variety of test programs.